CS 111 Operating Systems

Week 1 Discussion (01/10/2020)

Things we shall discuss:

- About the class
- Project 0 Prerequisites
  - VPN, SSH
- File Descriptors
  - What are they?
  - Different types of File Descriptors
    - POSIX file operations
    - Strerror
    - Getopt
    - Gdb
    - Demo on getopt and gdb
- -Wall, -Wextra
  - Demo
- Linux File Permissions
  - Demo
- Error Handling
  - Demo

About the class:

- **Split up:**
  - Lab0: 5%, Lab1-4: 10%
  - Midterm: 24%
  - Final: 30%
  - Class Evaluation: 1%
- **One Project per week** in this order: P0, P1A, P1B, P4A, P2A, P2B, P4B, P3A, P3B, P4C.
- **Due Date:** Projects will be due on Wednesdays at 11.59PM
- **Late Policy:** The lateness policy is exponential: 2^{N-1}, where N is the number of late days. This means:
  - a 1% penalty for a day late
  - - a 2% penalty for 2 days late
  - - 4% for 3 days
  - - 8% for 4 days
  - - 16% for 5 days
  - Submitting 1 minute late incurs a one day late penalty. Give yourself some time to submit.
- **Slip days**: There are no slip days. In ALL future project specs, IGNORE ANY MENTION OF SLIP DAYS!

About the class:

1. Please ask your questions related to assignments on piazza instead of sending emails to TAs, so that everyone can see the question and the answer. Before asking a question, check if someone has already asked that question. We will not answer repeated question.
2. Read the manual pages for all the functions you are going to use before writing the code.
3. Questions about your project grades should be addressed to the TAs responsible for it.
    a. Howard and Shaan: Lab 0, 1A, 1B and 4A
    b. Rishab and Nikita: Lab 2A, 2B and 4C
    c. Karen and Alex: Lab 3A, 3B and 4B

About the class:

4. Please remember to download your submissions and check if you submitted the correct files. Empty submissions or submissions in the wrong format cannot be graded and will therefore be scored with a 0. Therefore, check that your submission is in the right format (.tar.gz), and that the correct version of your files are included. Please pay extreme attention to your submission's filename, as names that do not follow the spec will receive a 15 point deduction.

5. Come to discussion after reading the spec at http://web.cs.ucla.edu/~harryxu/courses/111/winter20/ProjectGuide/index.htm

Project 0 Prerequisites

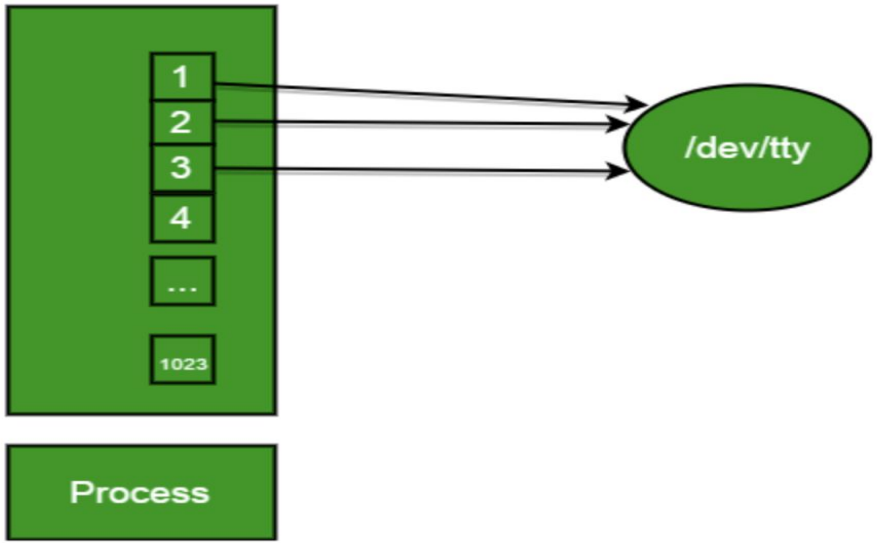
- For access to the linux machines, you would require to be connected to UCLA's network (either directly or through VPN)
- For VPN: https://www.it.ucla.edu/bol/services/virtual-private-network-vpn-clients
- For connecting to the Linux machines:
  - SEASNET Account:
    - If you don't have a SEASNET Account, apply for a seasnet account at https://www.seas.ucla.edu/acctapp/
    - If you already have a SEASNET Account, follow this: https://www.seasnet.ucla.edu/lnxsrv/
  - Open your terminal and type in:
    - ssh <username>@lnxsrv09.seas.ucla.edu

- Make sure you only test your code on that server. Please keep in mind that a submission compiles and runs on Mac/Ubuntu/Windows does not mean it works on lnxsrv09.

- **Setting PATH variable**:
  - There are different GCC versions on the SEASnet server. Please use the one in /usr/local/cs/bin. So specifically, please put **/usr/local/cs/bin** at the **beginning** of your **PATH**. Pay attention that each time you change your PATH, after you log out, the change will be restored, so you will have to find a way to overcome this issue.

File descriptors

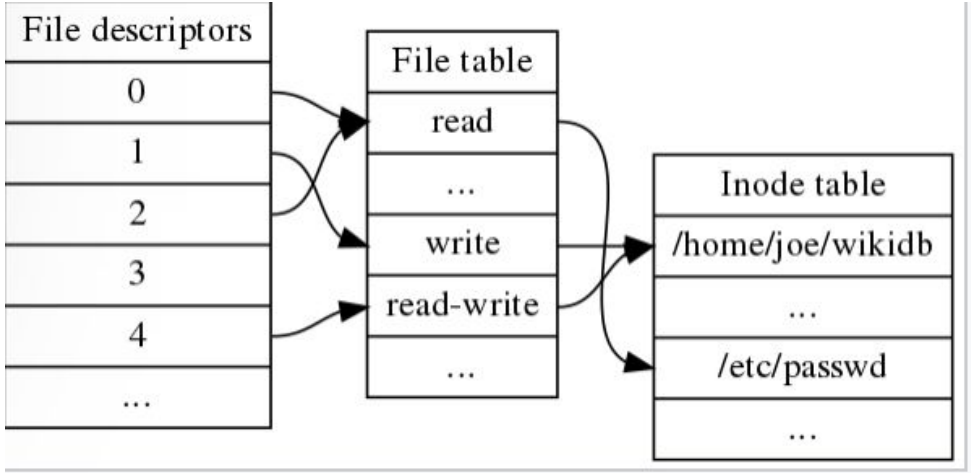
- **File descriptor** is integer that uniquely identifies an open file of the process
- **File Descriptor table**: File descriptor table is the collection of integer array indices that are file descriptors in which elements are pointers to file table entries. One unique file descriptors table is provided in operating system for each process.
- **Standard File Descriptors**: When any process starts, then that process file descriptors table's fd(file descriptor) 0, 1, 2 open automatically



- **Read from stdin => read from fd 0** : Whenever we write any character from keyboard, it read from stdin through fd 0 and save to file named /dev/tty.
- **Write to stdout => write to fd 1** : Whenever we see any output to the video screen, it’s from the file named /dev/tty and written to stdout in screen through fd 1.
- **Write to stderr => write to fd 2** : We see any error to the video screen, it is also from that file write to stderr in screen through fd 2.

Summary of File Descriptors

- Kernel maintains for each process **File Descriptor Table** which accesses a system wide table called the **File Table** (has modes/permissions like read, write, etc.), which in turn access the **Inode Table** which has the actual underlying files.



- Note that multiple file descriptors can refer to the same file table entry (e.g., as a result of the dup system call and that multiple file table entries can in turn refer to the same inode)

Example: input redirection

```
int ifd = open(newfile, O_RDONLY);
if (ifd >= 0) {
        close(0);
        dup(ifd);
        close(ifd);
}
```

Example: output redirection

```
int ofd = creat(newfile, 0666);
if (ofd >= 0) {
        close(1);
        dup(ofd);
        close(ofd);
}
```

Let's dive into the different types of FD (Reference: https://linux.die.net/man/2/, https://www.geeksforgeeks.org/input-output-system-calls-c-create-open-close-read-write/)

1. POSIX file operations
    a. open()
        i. **Function Signature**: int open(const char *pathname, int flags)
        ii. The flags can be argument flags, or creation flags or file status flags
        iii. The **argument** *flags* must include one of the following access modes: O_RDONLY, O_WRONLY, or O_RDWR. These request opening the file read-only, write-only, or read/write, respectively.

        iv. Example of a **creation** flag is **O_CREAT** (if the file does not exist it will be created)
        v. The **open**() system call opens the file specified by *pathname* and some optional flags.
        vi. The return value of **open**() is a file descriptor, a **small, nonnegative integer** that is used in subsequent system calls (read(2), write(2), lseek(2), fcntl(2), etc.) to refer to the open file.
    b. creat()
        i. **Function Signature:** int creat(const char *pathname, mode_t mode);
        ii. Mode : 1 is execute, 2 is write, 4 is read, and you add these together to get a permission set for a given class (e.g. group)
            1. 0001 is --------x
            2. 0002 is -------w-
            3. 0004 is ------r--
            4. 0007 is ------rwx

1. POSIX file operations
    a. open()
    b. creat()
        i. **open**() and **creat**() return the new file descriptor, or -1 if an error occurred (in which case, *errno* is set appropriately). Now these file descriptor's can be used to read(),close(), write().
    c. read()
        i. **Function Signature**: ssize_t read(int fd, void *buf, size_t count):
            1. Size_t represents the size of an allocated block of memory.
            2. Ssize_t represents signed size_t
            3. Refer: https://jameshfisher.com/2017/02/22/ssize_t/
        ii. read() used to read from a file descriptor
        iii. read() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.
        iv. The read operation commences at the current file offset, and the file offset is incremented by the number of bytes read.
    d. close()
        i. **Function Signature**: int close(int fd);
        ii. close() closes a file descriptor, so that it no longer refers to any file and may be reused.
    e. write()
        i. **Function Signature:** ssize_t write(int *fd*, const void **buf*, size_t *count*);
        ii. write() writes up to *count* bytes from the buffer pointed *buf* to the file referred to by the file descriptor *fd*.

2. Strerror: The strerror() function maps the number in errnum to a message string. Typically, the values for errnum come from errno, but strerror() shall map any value of type int to a message.

```c
char *strerror(int errnum)
{
    /*
        That is actually interpreting the standard by the letter, not intent.
        We only know about the "C" locale, no more. That's the only mandatory locale anyway.
    */
    return errnum ? "There was an error, but I didn't crash yet!" : "No error.";
}
```

3. getopt()

- The **getopt()** function is a built-in function in C and is used to parse command line arguments.
- Generally, the **getopt()** function is called from inside of a loop's conditional statement. The loop terminates when the getopt() function returns -1. A switch statement is then executed with the value returned by getopt() function.
- Try this out : https://docs.google.com/document/d/19fPdM_GtmG6Zjmku19ZFAgWm5TA0HKcMtOx9iJJUzAo/edit?usp=sharing

- **Function Signature**: getopt(int argc, char *const argv[], const char *optstring)

- **optstring** is simply a list of characters, each representing a single character option.
- **Return Value**: The getopt() function returns different values:
    - If the option takes a value, that value is pointer to the external variable **optarg.**
    - ‘-1’ if there are no more options to process.
    - ‘?’ when there is an unrecognized option and it stores into external variable **optopt**

4. gdb

- “GNU Debugger” A debugger for several languages, including C and C++
- It allows you to inspect what the program is doing at a certain point during execution.
- Errors like segmentation faults may be easier to find with the help of gdb.
- Reference:

  https://www.cs.umd.edu/~srhuang/teaching/cmsc212/gdb-tutorial-handout.pdf

Additional step when compiling program

- Normally, you would compile a program like:

```
gcc [flags] <source files> -o <output file>
```

For example:

```
gcc -Wall -Werror -ansi -pedantic-errors prog1.c -o prog1.x
```

- Now you add a -g option to enable built-in debugging support (which gdb needs):

```
gcc [other flags] -g <source files> -o <output file>
```

For example:

```
gcc -Wall -Werror -ansi -pedantic-errors -g prog1.c -o prog1.x
```

Starting up gdb

Just try "gdb" or "gdb `prog1.x`." You'll get a prompt that looks like this:

```
(gdb)
```

If you didn't specify a program to debug, you'll have to load it in now:

```
(gdb) file prog1.x
```

Here, `prog1.x` is the program you want to load, and "file" is the command to load it.

Before we go any further

gdb has an interactive shell, much like the one you use as soon as you log into the linux grace machines. It can recall history with the arrow keys, auto-complete words (most of the time) with the TAB key, and has other nice features.

Tip

If you're ever confused about a command or just want more information, use the "`help`" command, with or without an argument:

```
(gdb) help [command]
```

You should get a nice description and maybe some more useful tidbits...

Running the program

To run the program, just use:

```
(gdb) run
```

This runs the program.

- If it has no serious problems (i.e. the normal program didn't get a segmentation fault, etc.), the program should run fine here too.
- If the program *did* have issues, then you (should) get some useful information like the line number where it crashed, and parameters to the function that caused the error:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400524 in sum_array_region (arr=0x7fffc902a270, r1=2, c1=5,
r2=4, c2=6) at sum-array-region2.c:12
```

So what if I have bugs?

Okay, so you've run it successfully. But you don't need `gdb` for that. What if the program *isn't* working?

Basic idea

Chances are if this is the case, you don't want to run the program without any stopping, breaking, etc. Otherwise, you'll just rush past the error and never find the root of the issue. So, you'll want to *step through* your code a bit at a time, until you arrive upon the error.

This brings us to the next set of commands...

Setting breakpoints

Breakpoints can be used to stop the program run in the middle, at a designated point. The simplest way is the command "`break`." This sets a breakpoint at a specified file-line pair:

```
(gdb) break file1.c:6
```

This sets a breakpoint at line 6, of `file1.c`. Now, **if** the program ever reaches that location when running, the program will pause and prompt you for another command.

Tip

You can set as many breakpoints as you want, and the program should stop execution if it reaches any of them.

More fun with breakpoints

You can also tell gdb to break at a particular function. Suppose you have a function `my_func`:

```
int my_func(int a, char *b);
```

You can break anytime this function is called:

```
(gdb) break my_func
```

Now what?

- Once you've set a breakpoint, you can try using the `run` command again. This time, it should stop where you tell it to (unless a fatal error occurs before reaching that point).
- You can proceed onto the next breakpoint by typing "`continue`" (Typing `run` again would restart the program from the beginning, which isn't very useful.)

```
(gdb) continue
```

- You can single-step (execute *just* the next line of code) by typing "`step`." This gives you really fine-grained control over how the program proceeds. You can do this a *lot*...

```
(gdb) step
```

Now what? (even more!)

- Similar to "`step`," the "`next`" command single-steps as well, except this one doesn't execute each line of a sub-routine, it just treats it as one instruction.

```
(gdb) next
```

Tip

Typing "`step`" or "`next`" a lot of times can be tedious. If you just press ENTER, gdb will repeat the same command you just gave it. You can do this a bunch of times.

Querying other aspects of the program

- So far you've learned how to interrupt program flow at fixed, specified points, and how to continue stepping line-by-line. However, sooner or later you're going to want to see things like *the values of variables*, etc. This *might* be useful in debugging. :)
- The `print` command prints the value of the variable specified, and `print/x` prints the value in hexadecimal:

```
(gdb) print my_var
(gdb) print/x my_var
```

Setting watchpoints

Whereas breakpoints interrupt the program at a particular line or function, watchpoints act on variables. They pause the program whenever a *watched* variable's value is modified. For example, the following `watch` command:

```
(gdb) watch my_var
```

Now, whenever `my_var`'s value is modified, the program will interrupt and print out the old and new values.

Tip

You may wonder how gdb determines which variable named `my_var` to `watch` if there is more than one declared in your program. The answer (perhaps unfortunately) is that it relies upon the variable's **scope**, relative to where you are in the program at the time of the watch. This just means that you have to remember the tricky nuances of scope and extent :(.

Example programs

- Some example files are found in `~/212public/gdb-examples/broken.c` on the linux grace machines.
- Contains several functions that each should cause a segmentation fault. (Try commenting out calls to all but one in `main()`)
- The errors may be easy, but try using `gdb` to inspect the code.

Other useful commands

- `backtrace` - produces a stack trace of the function calls that lead to a seg fault (should remind you of Java exceptions)
- `where` - same as `backtrace`; you can think of this version as working even when you're still in the middle of the program
- `finish` - runs until the current function is finished
- `delete` - deletes a specified breakpoint
- `info breakpoints` - shows information about all declared breakpoints

Look at sections 5 and 9 of the manual mentioned at the beginning of this tutorial to find other useful commands, or just try `help`.

Some gdb commands summarized + demo

- **Continue** – An action to take **in the** debugger that will **continue** execution until the next breakpoint **is** reached or the program exits.
- **Step** – An action to take **in the** debugger that will **step** over a given line
- **Break <line number>** is a breakpoint and can be used to stop the program run in the middle, at a designated point.
- The **print** command prints the value of the variable specified
- **Clear <line number>** to clear breakpoint at that line
- **watch <my_var>** : Whereas breakpoints interrupt the program at a particular line or function, **watchpoints** act on variables. They pause the program whenever a watched variable's value is modified.
- Let's debug a program! (Reference: https://www.cprogramming.com/gdb.html)

Importance of -Wall and -Wextra flags

- gcc -Wall option flag
- gcc -Wall enables all compiler's warning messages. This option should always be used, in order to generate better code.
- -Wextra: This enables some extra warning flags that are not enabled by -Wall.

Example

Write source file *myfile.c*:

```
// myfile.c
#include <stdio.h>

int main()
{
    printf("Program run!\n");
    int i=10;
}
```

Regular build of *myfile.c* gives no messages:

```
$ gcc myfile.c -o myfile
$
```

Build of *myfile.c* with -Wall:

```
$ gcc -Wall myfile.c -o myfile
myfile.c In function 'main':
myfile.c:6:6: warning: unused variable 'i'
myfile.c:7:1: warning: control reaches end of non-void function
$
```

Linux File Permissions

- Permission Groups
  - Each file and directory has three user based permission groups:
    - **owner** – The Owner permissions apply only the owner of the file or directory, they will not impact the actions of other users.
    - **group** – The Group permissions apply only to the group that has been assigned to the file or directory, they will not effect the actions of other users.
    - **all users** – The All Users permissions apply to all other users on the system, this is the permission group that you want to watch the most.

Linux File Permissions

- Permission Types
  - Each file or directory has three basic permission types:
    - **read** – The Read permission refers to a user's capability to read the contents of the file.
    - **write** – The Write permissions refer to a user's capability to write or modify a file or directory.
    - **execute** – The Execute permission affects a user's capability to execute a file or view the contents of a directory.

Linux File Permissions

Viewing Permissions

- You can view the permissions by checking the file or directory permissions in your favorite GUI File Manager (which I will not cover here) or by reviewing the output of the ***"ls -l"*** command while in the terminal and while working in the directory which contains the file or folder.
- The permission in the command line is displayed as: ***_rwxrwxrwx 1 owner:group***

1. User rights/Permissions
    1. The first character that I marked with an underscore is the special permission flag that can vary.
    2. The following set of three characters (rwx) is for the owner permissions.
    3. The second set of three characters (rwx) is for the Group permissions.
    4. The third set of three characters (rwx) is for the All Users permissions.
2. Following that grouping since the integer/number displays the number of hard links to the file.
3. The last piece is the Owner and Group assignment formatted as Owner:Group.

Linux File Permissions

- Modifying Permissions
  - When in the command line, the permissions are edited by using the command ***chmod***. You can assign the permissions explicitly or by using a binary reference as described below.
- Explicitly Defining Permissions
  - To explicitly define permissions you will need to reference the Permission Group and Permission Types. The Permission Groups used are:
    - **u** – Owner
    - **g** – Group
    - **o** – Others
    - **a** – All users
  - The potential Assignment Operators are + (plus) and – (minus); these are used to tell the system whether to add or remove the specific permissions. The Permission Types that are used are:
    - **r** – Read
    - **w** – Write
    - **x** – Execute

Linux File Permissions

So for an example, lets say I have a file named file1 that currently has the permissions set to **_rw_rw_rw,** which means that the owner, group and all users have read and write permission. Now we want to remove the read and write permissions from the all users group.

To make this modification you would invoke the command: ***chmod a-rw file1***

To add the permissions above you would invoke the command: ***chmod a+rw file1***

As you see, to grant those permissions, you would change the minus character to a plus to add those permissions.

Linux File Permissions

Using binary references to set permissions:

A sample permission string would be **chmod 640 file1**, which means that the owner has read and write permissions, the group has read permissions, and all other user have no rights to the file.

The first number represents Owner permission; the second represents Group permissions; and the last number represents permissions for all other users. The numbers are a binary representation of the rwx string.

- ***r** = 4*
- ***w** = 2*
- ***x** = 1*

You add the numbers to get the integer/number representing the permissions you wish to set. You will need to include the binary permissions for each of the three permission groups.

Test yourself: To permissions on file named 'temp' to read _*rwxr*_____, what would you enter?

Reference: https://www.linux.com/tutorials/understanding-linux-file-permissions/

Error Handling (Reference: https://www.tutorialspoint.com/cprogramming/c_error_handling.htm)

- Most of the C function calls return -1 or NULL in case of any error and set an error code errno.
- It is set as a global variable and indicates an error occurred during any function call.
- You can find various error codes defined in <error.h> header file.
- So a C programmer can check the returned values and can take appropriate action depending on the return value.
- It is a good practice, to set errno to 0 at the time of initializing a program. A value of 0 indicates that there is no error in the program.

The C programming language provides **perror**() and **strerror**() functions which can be used to display the text message associated with errno.

- The perror() function displays the string you pass to it, followed by a colon, a space, and then the textual representation of the current errno value.
- The strerror() function, which returns a pointer to the textual representation of the current errno value.

Error Handling

Program Exit Status

- It is a common practice to exit with a value of EXIT_SUCCESS in case of program coming out after a successful operation. Here, EXIT_SUCCESS is a macro and it is defined as 0.
- If you have an error condition in your program and you are coming out then you should exit with a status EXIT_FAILURE which is defined as -1.
- EXIT_SUCCESS and EXIT_FAILURE are macros and hold conventional status value for success and failure, respectively

Thank you!